AFRL-RI-RS-TR-2010-211

A HIGH-PERFORMANCE RECONFIGURABLE FABRIC FOR COGNITIVE INFORMATION PROCESSING

Cornell University

DECEMBER 2010

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2010-211 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/

THOMAS E. RENZ
Work Unit Manager

/s/

EDWARD J. JONES, Deputy Chief
Advanced Computing Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
DECEMBER 2010	Final Technical Report	July 2007 – September 2010

4. TITLE AND SUBTITLE

A HIGH PERFORMANCE RECONFIGURABLE FABRIC FOR COGNITIVE INFORMATION PROCESSING

5a. CONTRACT NUMBER
FA8750-07-2-0191

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
459T

6. AUTHOR(S)

Rajit Manohar

5d. PROJECT NUMBER
AC

5e. TASK NUMBER
CU

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Cornell University
373 Pine Tree Road
Ithaca NY 14850-2820

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2010-211

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. PA# 88 ABW-2010-6251
Date Cleared: 29 November 2010

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Cognitive systems have requirements that are not met by existing commercially available architectures such as multi-core microprocessors or reconfigurable logic. In collaboration with Air Force Research Laboratory (AFRL), this project had the goal of creating a hybrid architecture that combines an existing power efficient AFRL design with an asynchronous high performance reconfigurable fabric. A new reconfigurable fabric was created with a number of unique features, including support for low leakage operation via power gating, additional hardware support for dynamic reconfiguration, and features that enable virtualization of the configuration memory of the fabric. A chip was created in collaboration with AFRL that included an AFRL system design and the new asynchronous fabric and submitted for fabrication through the Trusted Foundry Access Program.

15. SUBJECT TERMS
Asynchronous Field Programmable Gate Array, High Performance Computing, Trusted Design, Fast FPGA, System on a Chip, Multi Node System

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			THOMAS E. RENZ
U	U	U	UU	27	19b. TELEPHONE NUMBER (*Include area code*) N/A

Table of Contents

List of Figures

1. Introduction

There are two major computation platforms that have survived the test of time when it comes to a combination of efficiency and programmability: the microprocessor, and the Field Programable Gate Array (FPGA). Microprocessors are designed to implement a von Neumann abstraction of computation, where the computation is specified using a sequence of instructions that specify how the state of the system is to be modified. High-level languages are translated into these instruction sequences using a compiler, hiding the complexity of the detailed instructions from a user of such a system. FPGAs are designed to implement hardware described using combinational logic and state holding elements. The hardware itself is described using a high-level hardware description language, and logic synthesis tools are used to translate the description into the appropriate configuration information for the FPGA. Both these approaches translate a static, functional description of a computation into a static, physical implementation that implements the computation.

Cognitive systems, on the other hand, exhibit properties that are more of a dynamic nature. They learn from their mistakes, and repetition makes them more proficient at performing a particular task. A microprocessor-based approach with storage that captures learnt information was a natural way to perform such computations, and this approach has been widely adopted by the artificial intelligence community. However, traditional cognitive algorithm implementations on conventional microprocessor-based platforms require large amounts of memory, and have high input/output and communication requirements. Mainstream industry has moved to multi-core platforms, but even these platforms do not support efficient cognitive computing [1].

The human brain provides an existence proof of a highly efficient platform for cognitive information processing. This platform is very different from current computing architectures. Salient features of the platform that are a departure from conventional thinking include the lack of separation between computation and storage, and the importance of connectivity between components that can dynamically evolve over time based on the computation being performed. These properties more closely resemble FPGAs, rather than conventional microprocessors.

Structural information about the way in which the computation is organized is not captured in a microprocessor-based approach. Another limitation comes from the observation that storage and computation are physically separate in a microprocessor, whereas all evidence points to integrated storage and computation in neural systems.

This report contains a summary of the work that was conducted in creating a platform that was more suitable for cognitive computing tasks. The platform consists of a hybrid between a microprocessor and FPGA, with additional features in the FPGA that support cognitive computing tasks. As part of this effort, the features introduced in the platform have properties suitable for implementing certain hardware security features as well.

2. Methods and Procedures

The Cornell Asynchronous Very Large Scale Integration (VLSI) and Architecture group previously developed a high-performance FPGA fabric for general-purpose computing. Compared to the state-of-the-art commercial FPGAs from industry, the performance of the fabric was three to five times higher—a significant improvement. Compared to the best previously developed asynchronous FPGAs, the Cornell FPGA was almost twenty times faster in terms of application throughput [2,3,4]. This dramatic performance increase makes the fabric ideally suited to be integrated into a system containing a high-performance microprocessor.

2.1. Asynchronous FPGA Concepts

In terms of the major building blocks, the asynchronous FPGA (AFPGA) architecture looks like a traditional synchronous island-style FPGA such as a Xilinx Virtex [5]. The FPGA contains a configurable logic block (LB), a configurable interconnect (SB), and connection boxes (CB) that are used to connect the interconnect to the logic blocks. Figure 1 shows a high-level view of a generic FPGA architecture.

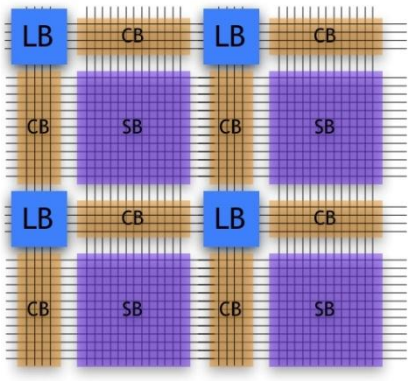

Figure 1. Island-style FPGA Architecture.

The major differentiating feature of the AFPGA versus a conventional FPGA architecture is the underlying computation model used to implement the configurable fabric. Instead of thinking of computation in terms of gates and registers, the AFPGA implements a computation specified by a dataflow graph [6]. In the dataflow graph model, computation is described by operations on data values or "tokens" flowing through the graph. Tokens correspond to valid data items being processed by elements of the dataflow graph. Nodes in the dataflow graph include function blocks that can perform computation, as well as routing elements for sending tokens to the appropriate destinations. Token arrival at a dataflow node can be thought of as an "event" that triggers activity in the AFPGA.

Dataflow computations can be implemented in a variety of ways. The AFPGA uses a set of basic building blocks for dataflow computations based on over ten years of experience designing both high-performance asynchronous microprocessors as well as low power asynchronous microprocessors. In designing these complex systems, it was found that there were only a few circuit topologies that led to efficient implementations. These are summarized below, and described briefly. The key building blocks are shown in Figure 2, and their functionality is summarized as follows:

- **Function**. The function block has N inputs and one output. This is the basic logic computation element. It receives a data token from each of its inputs, computes a function of the received input data, and produces the value as an output token.
- **Source**. A source produces a stream of constant tokens on its output.
- **Sink**. A sink consumes any tokens it may receive on its input.
- **Copy**. A copy is used to implement the equivalent of signal fanout. It replicates every input token it receives on all of its outputs.
- **Initial**. An initial block begins by producing a token on its output, and then after that simply copies any input token it receives to its output.
- **Merge**. The merge block is a conditional block. It receives a data token from its control input (shown as a horizontal arrow above). The value of this data token is used to select an input port. The input data on the selected input port (vertical arrows) will be sent to the output. No other input tokens are consumed.
- **Split**. The split block is the dual of a merge. It receives a data token from its control input (shown as a horizontal arrow above). The value of this data token is used to select an output port. The input data value (vertical arrow) will be sent to the selected output port.

Arbitrary computations can be constructed from these basic building blocks [6,7]. The logic block for the FPGA contains all the necessary components to implement any dataflow computation. In particular, it contains programmable implementations of all the dataflow elements shown above.

As an example, the function element is implemented using a programmable four-input lookup table (LUT). Such a dataflow LUT waits for valid data tokens to arrive on all of its inputs, and then produces a new data token on its output based on the truth-table configuration of the LUT. Other support logic such as fast carry-chains and multiplier

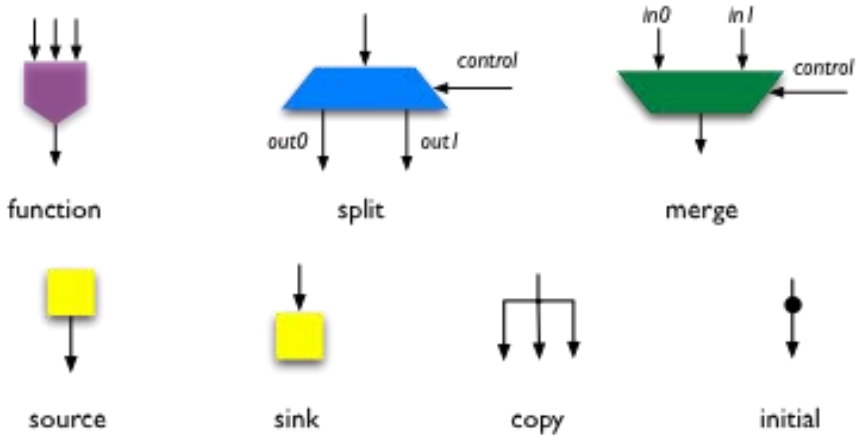

Figure 2. Building Blocks for Dataflow Computation.

support operate in a similar fashion.

The key performance amplifier in the AFPGA is its flexible routing network. A conventional FPGA has over 70% of its delay in the routing network [8]. Since the AFPGA operates using a dataflow model, pipeline stages corresponding to queues can be introduced into the routing network without impacting the correctness of the computation being performed by the AFPGA! This means that designs can benefit from pipelining without the additional cost required from electronic design automation (EDA) tools to support interconnect pipelining. In the first AFPGA implementation, pipelined stages were introduced in the switch boxes in the AFPGA interconnect [2,3].

The asynchronous nature of the fabric and the deeply pipelined implementation implies that only local paths limit AFPGA performance. For instance, even though the interconnect is configurable, it is impossible to create a path that contains a large number of switches as in a conventional FPGA because all such paths are partitioned by the presence of pipelining in the switch boxes. This is a dramatic difference from a conventional FPGA, where long paths through the routing network that have multiple switches and buffers are usually on the critical path [8]. The net effect is that the critical path limiting the peak frequency of the AFPGA corresponds to the lookup table access rather than the routing network.

The nature of the pipelined interconnect makes the entire AFPGA highly modular. In particular, because communication between components on the AFPGA uses the dataflow model, the delay of the communication link is not part of the interface specification. This enables a highly modular approach to the design of the AFPGA, where sub-blocks can be pre-placed without significantly impacting performance. Indeed, if data flow between one sub-block and another is unidirectional (as in a computation pipeline), there is no loss in throughput by using a modular approach to synthesis and place-and-route.

The impact of aggressive pipelining on the overall performance of the AFPGA is significant. In a 0.18µm feature size, the measured peak performance of the AFPGA architecture was 674 MHz. For reference, the baseline Xilinx architecture in a similar feature size performs at 240MHz [4]. More important, first pass synthesis results for a variety of benchmarks demonstrate robust performance. For example, a synthesized Finite Impulse Response (FIR) filter core would exhibit a performance of 75% of the peak performance of the AFPGA.

The only real source of performance loss in the AFPGA arises due to data-dependent loops in the computation graph. To illustrate this, consider a loop in a synchronous circuit that has a logic depth of d seconds, and contains k registers. The design cannot operate at a throughput that exceeds k/d, simply because there aren't a sufficient amount of registers on the loop to exceed that performance. This is a fundamental or "algorithmic" limit of the design. The AFPGA is constrained by this limit, and we have found that in practice the AFPGA can achieve a performance that is close to 80% of the algorithmic limit or higher.

2.2. Asynchronous FPGA Software

In addition to hardware, an FPGA platform requires significant software development. In particular, design automation tools that take computations described in some language, map them to the FPGA architecture, and then generate the configuration bits required for the FPGA must be provided for an FPGA to be usable. This task is broken down into three major steps.

Logic Synthesis. The first step in the software flow is logic synthesis. In this step, arbitrary computations are specified in a language that resembles a concurrent version of the C programming language. This description is converted into a dataflow implementation using a new compiler representation called "static token form." Wide datapath operations are broken down into bit-level operations, and various standard logic optimizations are performed on the dataflow graph.

Logic Packing. In this step, the dataflow elements are clustered to match the logic structure in the programmable logic blocks in the asynchronous fabric. This improves the logic utilization of the overall design.

Place and Route. For placement and routing, an open-source place-and-route tool known as "`vpr`" (for versatile place and route) developed by the University of Toronto can be used. The output of this process is converted into a configuration file to match the interconnect topology of the AFPGA architecture.

The key new step in logic synthesis is illustrated below, by describing a simple computation and how it is implemented using a dataflow graph. As a first step, consider straight-line programs that do not have any conditional execution. The program below shows a simple computation where two different functions are computed.

```
a:=RECV(A);  b:=RECV(B);   x:=f(a,b);
 SEND(x,X);   x:=g(a,b);   SEND(x,Y);
```

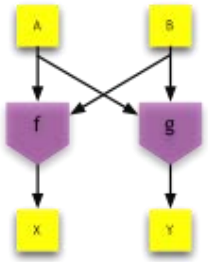

Figure 3. Dataflow translation of a straight-line program.

The inputs are received on ports `A` and `B`, and two values `f(a,b)` and `g(a,b)` are transmitted on output ports `X` and `Y` respectively. The static token form representation was used as a systematic way to translate the computation described in a high-level language into a dataflow graph. This representation matches the requirements for hardware implementation of dataflow graphs, and the use of this representation provides a simple mechanism for mapping designs to dataflow elements. This approach has been automated, and the net effect is the creation of the dataflow graph shown in Figure 3 [9].

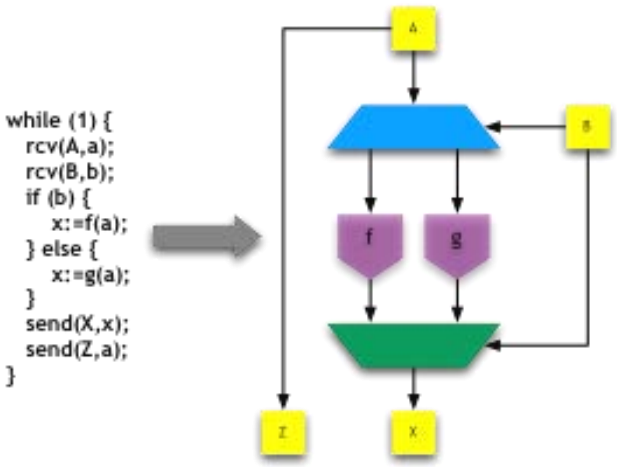

Figure 4. Translation of conditional execution into a dataflow graph.

While in this particular case, the dataflow graph may be "obvious", it should be stressed that the static token form approach is a *systematic* and *automated* method to generate this graph from the computation.

The translation is more complicated when we have conditional execution. Figure 4 below shows an example of a program with an if-statement and its corresponding dataflow graph. In this case, the value of `a` is used either to compute `f(a)` or `g(a)` depending on the value of another variable `b`. To implement this functionality, special split and merge dataflow elements are introduced. These elements are used to route the data to the appropriate function, and then to collect the result and transmit it on the primary output of the block. The dataflow building blocks in Figure 2 are sufficiently expressive so that any program can be implemented using those blocks.

The Defense and Advanced Research Projects Agency (DARPA) supported the work described above as part of the Architectures for Cognitive Information Processing (ACIP) program [10]. The research results provided an improvement in programmable logic performance that was sufficient to launch a commercial enterprise, namely Achronix Semiconductor Corporation.

The work described above provides the technology basis for a high-performance FPGA fabric. The goal of the funded effort was to transition this technology into an Air Force Research Lab (AFRL) design to be fabricated through the Trusted Foundry program. Although the basic principles of operation of the AFPGA were known, there were a number of changes required to meet AFRL requirements and for cognitive computing in general.

2.3. Dynamic Reconfiguration with Meta Bits

The AFPGA was designed as a static architecture, just like most commercial FPGA designs. However, dynamic re-configurability was a requirement for two major reasons: (i) FPGAs have significant area overhead due to their flexibility [11]. Even the largest commercially available FPGAs are too small to accommodate sophisticated software applications. It is important to be able to time-multiplex the FPGA area for different functions based on evolving application requirements. (ii) Dynamic re-wiring seems to be an important property of cognitive architectures. It was important for the AFPGA to be able to support this functionality. There are a number of technical difficulties that arise

when creating an AFPGA architecture whose configuration can be dynamically changed. These arise due to two kinds of dynamic configuration changes: (i) Using available real-estate for new computations; (ii) Modifying existing real-estate based on new information.

Configuration information in an AFPGA can be treated as a large on-chip memory block. Executing write operations to the configuration memory of the AFPGA can modify bits of configuration. Adding additional mapped designs to the AFPGA *while part of the AFPGA is being used for a different function* is a challenging task, because it requires fine-grained addressability of all the configuration bits. Unfortunately, introducing bit-level addressing capability for configuration memory is impractical because it has a large impact on the area of the AFPGA design.

The method used to eliminate the area overhead was to introduce a set of *meta-configuration* bits throughout the AFPGA fabric. These meta-configuration bits can be used to write-protect parts of the AFPGA configuration memory. The meta-configuration bits are designed so that they control access to configuration information for logically coherent sections of the AFPGA fabric. For example, one bit controls access to all the configuration information required for LUTs in a logic block since it is expected that these LUTs would be grouped together for purposes of reconfiguration. Meta-configuration bits can also be used to partially modify an existing configuration. Once again, the meta-bits are used to select the section of the design to be updated. The bits serve as a selection mask at a granularity that is appropriate for AFPGA reconfiguration.

The introduction of meta configuration bits enables both dynamic configuration modalities, and the overall area impact of this additional information was found to be approximately 1%.

2.4. Synchronous Interface Design

An AFPGA expects asynchronous protocols on its inputs and outputs. However, most complex designs implemented with commercial design automation tools are clocked. One of the requirements for the AFRL project was to create a clocked interface to the AFPGA architecture.

The method used for interface purposes was to modify the traditional input-output (I/O) interface ring of the AFPGA with synchronous register-based ports. These I/O ports can be configured as special registers that can be externally read (output registers) or written (input registers) by the processor. When an input register is written by the external entity, the value stored in the register is injected into the AFPGA fabric as a dataflow token; when an output register is read by the external entity, the current value of the output register is transferred to the external entity and the value of the output register can be updated by the AFPGA on the following cycle.

The input register behaves as a one-place queue, and because the AFPGA fabric is fast and heavily pipelined, the expectation is that there will be no stalls during typical AFPGA operations. The output register also behaves like a queue, but it has an extra bit

(visible as a separate special register) to indicate if there is valid data present or not. The external entity can either check this bit and wait for it to be valid before reading the output register, or know (by timing analysis of the logic mapped to the AFPGA) when it is safe to read the output relative to the time at which the input was provided. On a read of the output register (assuming the valid bit was set), the valid bit is automatically cleared, and only then can the output register be updated with the next value from the AFPGA. Since the pipelined AFPGA fabric implicitly queues data throughout the fabric, it is expected that no additional queue space will be necessary for the input/output registers. The input/output registers and valid bits are visible externally, and can be read or written in the usual way. These registers also contain a conventional synchronizer circuit to hide the asynchronous/synchronous boundary [12, 13].

These interfaces were designed in collaboration with several researchers at AFRL, so as to ensure that the AFPGA could communicate with the AFRL design in a manner that met the requirements for the larger AFRL project.

2.5. Tool Chain for Design Mapping

Previous work in developing a software tool chain for mapping designs to the AFPGA was based on translating a high-level asynchronous design entry language into a configuration bit-stream [8]. This tool flow, while efficient, uses a design entry language that is not used outside the asynchronous design community. To make the AFPGA accessible to hardware designers that have experience with commercially available design tools, the project also created a new tool flow based on commercial design entry languages such as Verilog and VHDL (Very High Speed Integrated Circuit Hardware Description Language).

Commercial tools for logic synthesis create a clocked gate-level netlist from a Verilog or VHDL description. Companies such as Synopsys, Mentor Graphics, and Cadence have developed these tools over decades. The method adopted by this project was to leverage this investment and convert the output of commercial synthesis tools into a form suitable for mapping to the AFPGA architecture.

The design flow for the new tool chain operates as follows:

- Synthesize the Verilog or VHDL description using a commercial synthesis tool.
- Use the open source `abc` package to map the output of the synthesis tool into look-up tables of a fixed size (4-input lookup tables for the AFPGA).
- Clean up the output of `abc` using a simple line-based script.
- Use the open source `vpr` package to place and route the AFPGA.
- Translate the output of `vpr` into the correct value of all the configuration bits for the AFPGA.

This approach leverages commercial synthesis tools and existing designs described using popular commercial hardware description languages [14].

2.6. Partially Automated Physical Implementation

The implementation of the AFPGA required significant effort from a physical design standpoint. Cornell has developed a number of tools for the logical implementation of high-performance asynchronous logic circuits. These tools were leveraged in the creation of the new AFPGA architecture.

The tools use a hierarchical, type-safe design entry language called "ACT" (for Asynchronous Circuit Tools). The design entry language uses an abstract representation of transistor pull-up and pull-down networks to specify every gate in the circuit that implements the AFPGA. Transistor sizes can also be specified in the language, and the entire language is automatically converted into a Simulation Program with Integrated Circuit Emphasis (SPICE) format for both transistor-level simulation and physical layout.

To simplify the physical layout procedure, part of the work conducted on this project was to develop a tool called "`pandR`" (for "place and route") whose input is in SPICE format. This tool is user-guided, and contains support for automating some of the tedious manual layout tasks. The tool is also aware of the physical design rules for the technology, and respects these design rules to the extent possible. This tool was used to accelerate the physical implementation process. In addition, standard layout editors were used to complete the physical layout.

3. Results and Discussion

The physical design of the AFPGA layout was completed in IBM's 65nm LP process (10LP) available through the Trusted Foundry Access Program. The AFPGA was designed as an intellectual property (IP) block with the appropriate design files so that it could be integrated with AFRL's system design in the same way as third-party IP blocks such as phased locked loops, high-speed serial interfaces, memories, etc. are integrated in commercial design flows. AFRL submitted the chip for fabrication through the Trusted Foundry Access Program.

3.1. New AFPGA Core Architecture

Based on prior AFPGA design experience, a number of changes were made to the architecture of the AFPGA logic block. These changes were accompanied by corresponding changes to the software tool flow to support the new logic block architecture.

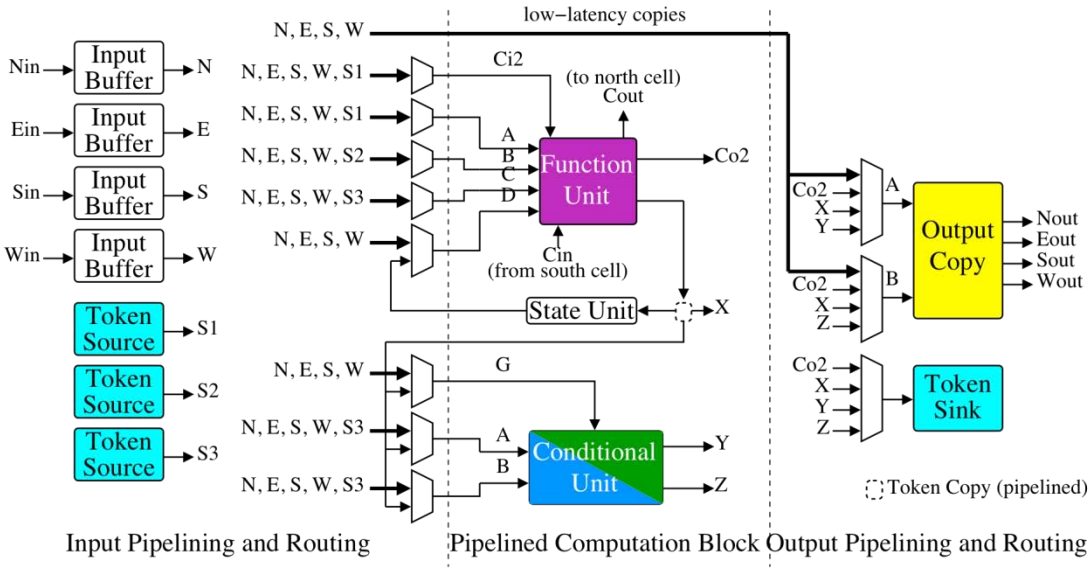

Figure 5. Original AFPGA Logic Block.

The original AFPGA resembled the logic block architecture of a Xilinx Virtex series FPGA [5]. The core computation block was a four-input lookup table, which has previously been shown to be a good compromise between performance and area-efficiency. The logic block also contained hardware support for fast ripple carry adders via both additional logic and a dedicated routing path for fast carry chains that bypassed the general-purpose routing network. In addition, multiplier support was provided through a programmable AND gate. Figure 5 shows the detailed block diagram of the original AFPGA architecture [3].

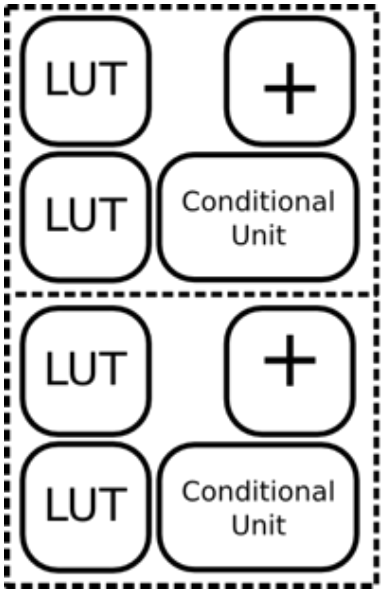

Figure 6. Logical configuration of the core of the new AFPGA logic block.

There were a number of limitations of this architecture that were remedied in the new AFPGA design. First, the logic block architecture in Figure 5 contains a single four-input lookup table. While this was suitable for a first-generation AFPGA, modern reconfigurable fabrics have clustered logic blocks containing multiple lookup tables per logic block. This choice results in significantly reduced routing requirements, as larger sections of logic can be mapped to a single logic block without resorting to the global routing network. The result of this analysis was that our new AFPGA architecture contains four four-input lookup tables per logic block. A second major change was made in the design of the function unit. In the original AFPGA (as in conventional synchronous FPGAs from Xilinx/Altera) the lookup table that is part of the function unit is re-used to implement a

ripple-carry adder by the introduction of some support logic (exclusive OR gates). Instead of this, our new AFPGA design uses dedicated hardware adder structures that share inputs with the lookup table. The logic block can either be configured to use the lookup table, or configured as an adder. The benefit of this approach was that carry chains in our architecture are twice as fast, because the adder block can be designed to perform two bit additions in a single stage rather than just a one-bit addition. Figure 6 provides a logical view of the new AFPGA logic block core. Each block can be thought of as two halves, each containing two four-input lookup tables, a two-bit adder, and a conditional unit containing the split and merge dataflow elements.

The internal connectivity supported by the logic block is significantly enhanced compared to the original AFPGA architecture. Inputs arriving from the switch box are connected to the input connection box. This connection box provides simple point-to-point connectivity that allows inputs to be connected to a small number of dedicated input channels for the overall logic. This is followed by an input cross-bar with copying support, which converts the small number of inputs from the switch box into a set of inputs for each half of the AFPGA logic core (Figure 6). Copying support in this cross-bar allows a single input from the routing fabric to be copied to multiple input pins of the logic core. After passing through the logic core, the output of the logic block can also be copied to multiple destinations by the output copying block, and this is followed by the output connection box that provides connectivity back to the global routing network. Figure 7 is a block diagram that describes the overall logic block with the local routing network support. Another interesting feature of the new AFPGA architecture is the generality of feedback connections that are available. Outputs from the logic core can be fed back into the input cross-bar, enabling general connectivity within the logic core itself. Finally, the logic block contains a number of buffers that are used to decouple different parts of the design and increase throughput through pipelining.

Each AFPGA tile also contains meta-configuration bits that can be used to protect various parts of the configuration space of the tile. In the completed version of the AFPGA, the configuration architecture assumes that each logic block can only be configured for one function; on a change, either no elements within a logic block are impacted or all the elements in the logic block are impacted. This simplification reduces

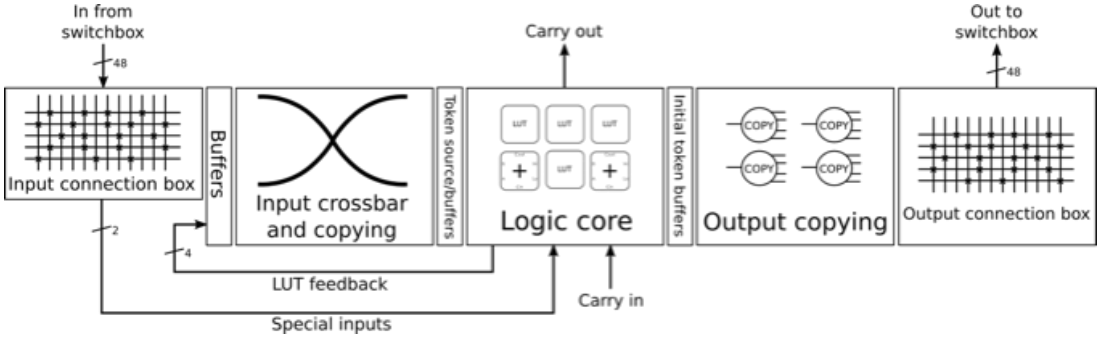

Figure 7. Internal connectivity provided by the AFPGA logic block.

the meta-configuration bit overhead, keeping it at ≈1% when both the configuration bits for routing and logic are taken into account.

3.2. Dynamic Reconfiguration Ability

The new AFPGA routing network contains significant support for dynamic reconfiguration. Apart from the per-logic block configuration support described above, an important consideration in the design of the AFPGA was configuration support for routing. Meta-configuration bits were designed to allow a small group of tracks to be re-configured without impacting existing configurations of adjacent tracks.

Figure 8 provides an illustration of the reconfiguration support provided in the new AFPGA. The figure on the left corresponds to a configuration where the blocks highlighted in green correspond to some computation mapped to the AFPGA, and the blue path is configured to connect a source to a destination. The new configuration preserves the configuration in green, but changes the source-destination path to the one shown on the right. The meta-configuration bits are organized so as to support this level of dynamic reconfiguration. Observe that the new path occupied by the blue routing track utilizes resources that are physically adjacent to the green configuration. This is made possible by the meta-configuration bits, as only the configuration corresponding to the blue path is modified.

The value of the meta configuration bits are a global mask which identifies the parts of the AFPGA fabric impacted by reconfiguration. The key difference between meta configuration bits and other possible mechanisms (e.g. using bit-addressable configuration memory) is that the meta bits group configuration information into logical units that are meaningful from the standpoint of dynamic reconfiguration. Meta-configuration bits are also grouped into words of a size that corresponds to the normal granularity of the configuration memory organization. By writing the appropriate meta-

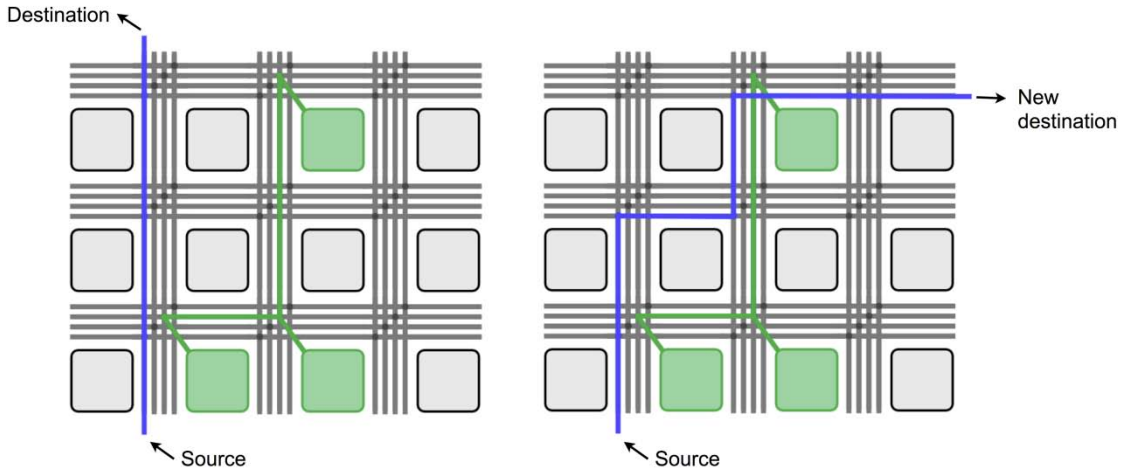

Figure 8. An example of supported routing re-configuration in the AFPGA.

configuration bits, the appropriate subsets of ordinary configuration bits are selected for modification. The update pictorially depicted in Figure 8 would proceed as follows:

- Step 1. All meta-configuration bits would be cleared, thereby preventing any configuration modifications.
- Step 2. The meta-configuration bits corresponding to the original and new blue route are set, thereby enabling modifications to the configuration bits controlling all the gates impacted by the route change.
- Step 3. The configuration bits are updated. While full configuration word writes are attempted, the masking due to meta-configuration bit values ensures that only the appropriate subset of configuration information is modified.
- Step 4. The route is initialized by following the reset protocol for the asynchronous logic. Once again, meta configuration bits are used to ensure that only the new route is initialized. The green configuration is not impacted by the reset protocol.

3.3. AFPGA Configuration Memory and Security

The previous AFPGA design used a shift register chain to implement the configuration memory. Unfortunately, a shift register chain is very inefficient from the standpoint of area. The new AFPGA architecture uses a modified static memory cell as the core configuration storage element to minimize the amount of area required for configuration bits. This is similar to the approach taken in commercial FPGA implementations [8].

There are a number of unique challenges when designing a configuration memory cell. It may seem that a standard static random access memory (SRAM) can serve the function of configuration memory. However, there are a number of requirements for a configuration memory storage element that differ from those of a standard SRAM bit cell.

A traditional SRAM bit cell has to operate correctly only in an array environment. Reading and writing the bit is performed through shared lines, and hence the capacitive loading on various access ports for a standard SRAM is well-defined and deterministic. Apart from edge effects (that sometimes require dummy SRAM cells to surround the core memory array), bit-cells have an identical—or at least very predictable—environment.

FPGA configuration memory is used to control gate voltages on pass transistors. This control voltage is required *at all times*—which means that all the bits of configuration memory need to be accessed continuously, and in parallel. To simplify the design of the AFPGA configuration memory, isolation buffers were introduced between the stored configuration bit and the pass transistor gates so that a dense configuration memory bit cell would have a deterministic electrical environment. Wiring considerations imply that area-optimized FPGA layout places configuration memory bits near the locations where they are used. Therefore it is impractical to create large, dense configuration memory

arrays—since each bit needs at the very least a wire that connects it to the appropriate pass transistor gate. The AFPGA configuration memory was organized in very small arrays that were placed near the locations where their values were needed. This observation also means that the configuration memory bits cannot use high-density foundry SRAM cells, because foundry cells require that the bits be organized in a dense array configuration.

The AFPGA uses approximately 1Kbit of configuration memory per four input lookup table. Of this total storage requirement, only sixteen bits are required for the lookup table configuration; the rest correspond to the average cost of configuration information for routing. Even a modest AFPGA size (e.g. 1024 LUTs) requires a configuration memory of the order of 1 Mbit! However, the bits of this memory are not organized in a densely packed configuration as discussed earlier, but look more like a sparse array that logically behaves as a normal SRAM. The result is that the AFPGA configuration memory requires much longer bit lines and word lines compared to a traditional SRAM that has the same total storage.

To handle the cross-talk and coupling issues created by the long word and bit lines, the AFPGA configuration memory is partitioned into smaller banks, with local drivers placed periodically throughout the AFPGA array. Critical parameters that govern the read and write margins in the AFPGA configuration memory are therefore isolated to a small physical region sized to maximize robustness for a reasonable area budget.

Meta configuration bits are also organized in the same manner as ordinary configuration bits. A fixed section of the address space of the configuration memory was allocated for meta-configuration information. Operations on meta-configuration bits were identified by comparing a subset of the address bits to a fixed value corresponding to the meta-configuration address space.

Reads and writes to ordinary configuration bits were *gated* by the appropriate meta-configuration bits. Each AFPGA tile contains meta-configuration bits that control ordinary configuration bits within the tile. The overall area overhead of this scheme was found to be ≈1%. Reads and writes to configuration bits that are not enabled (because their meta-configuration bits are clear) are ignored; this effectively write-protects any configuration bit that is protected through the meta-configuration information.

Since meta-configuration bit addresses are easily identified externally, memory address protection mechanisms can be used to protect not just meta-configuration bits but ordinary configuration bits. To write-protect any AFPGA configuration, the following steps can be followed:

1. Set the meta-configuration bits so that reads and writes to the appropriate ordinary configuration bits are disabled.

2. Place the system in a mode that disables any modifications to the meta-configuration bits.

At this point, any writes to AFPGA configuration memory will not modify the protected configuration information. Write protection of the meta-configuration bits is supported by

a single input signal to the AFPGA array. Reading the configuration memory can also be disabled by another input pin to the AFPGA. By disabling reads and writes through a secure path, the current configuration of the AFPGA can be protected from an external attacker.

3.4. AFPGA Interface Circuits

Integrating an asynchronous FPGA with standard synchronous logic requires an interface circuit to manage the potential metastability that can arise when an asynchronous signal change violates setup/hold time requirements of the clocked state-holding element.

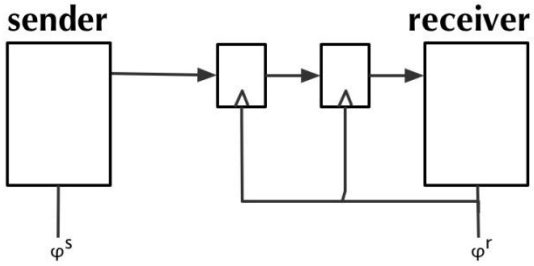

Figure 9. Basic two-flop synchronizer.

The circuit used for the AFPGA designed for this project was a variant of a conventional two-flop synchronizer [12]. The basic principle behind a two-flop synchronizer is shown in Figure 9. A signal from the sender passes through two flip-flops that are clocked by the receiver. The first flip-flop could have its setup or hold time violated, and so its output could be metastable. However, this metastable state has an entire clock cycle to resolve itself before being examined by the receiver [13]. This approach provided sufficient margin for the AFRL design given the operating frequencies and underlying process technology.

Another requirement of a synchronization interface that must be ensured is that multiple values cannot be transmitted while the synchronizer is in a metastable state. This particular problem was handled by providing full flow control between the sender and receiver in the circuits designed for the AFPGA [12].

3.5. Circuit Changes

The AFPGA circuits were updated and ported to the 65nm process technology used for this project. These changes involved evaluating analog properties of the asynchronous circuits to ensure that the circuits continued to conform to the digital abstraction. This evaluation impacted the overall transistor sizes of the circuits, and in some cases required re-design to ensure that large loads were driven with the appropriate circuit family.

There were two major changes that were made to the circuits in the AFPGA. First, the lookup table logic was modified so that its inputs used one of four encoding rather than dual rail encoding. This reduced the number of transistors in series required to implement the logic stack for the lookup table, and improved the power and performance of the

circuit. The second major change was more global, and involved the introduction of power gating transistors in the AFPGA architecture.

Power gating was introduced as a means to control the leakage current when sections of the AFPGA are unused. Since configuration information determines which parts of the AFPGA circuit are in fact active, additional configuration bits were used to control power-gating transistors connected to the power supplies for different sections of the AFPGA. Fine grained power gating enables small clusters of logic to be gated off, thereby providing leakage management when the AFPGA is underutilized. Note that power gating transistors provide another mechanism for disabling AFPGA functionality. Hence, the configuration bits used to control power gating must also be protected when a configuration must be locked down.

3.6. Physical Implementation

The physical design of the AFPGA was undertaken using a combination of commercially available layout editors as well as a custom-designed tool to automate some of the physical design process. The newly developed "layout assistant", called "`pandR`" (for place and route), was created to simplify some of the aspects of creating the physical layout for the AFPGA.

The input to `pandR` is a SPICE netlist that contains transistors and their widths and lengths. The input is hierarchical, and the input hierarchy is used to create the layout hierarchy. The other input to the tool is a technology description file that contains all the necessary information required by the tool to create design rule clean layout.

On startup, the tool creates a hierarchical graph representation of the layout that has to be generated. Transistors are grouped into *stacks* that correspond to contiguous regions of diffusion in the physical layout. Each stack is assigned a default position, and they are grouped based on the gate they correspond to and the type of transistor they contain (p-type or n-type).

The placement algorithm used is a variant of quadratic placement with the goal of minimizing the potential energy of the stacks [15]. Unlike traditional physical automation tools that create a placement for all the layout elements and then follow that by routing all the wires, `pandR` routes partial nets while placement is being performed.

The main loop that the tool executes can be described as follows:

1. Build a matrix for the quadratic solver, including all previously placed stacks, unplaced stacks, and I/O pins.

2. Solve the quadratic equation and assign all unplaced stacks a position.

3. Pick a stack to place, and legalize its position so that all design rules are satisfied.

4. Run the router to draw wires that may have been erased due to previous steps.

This continues until either all stacks have been placed, or until the tool cannot complete a step without violating design rules.

Figure 10 shows a screen shot of the layout assistant for a simple circuit used to implement a queue. One of the more unique features of this tool is that the I/O pins (shown as black circles) of a circuit can be placed in the interior of the layout region rather than just on the boundary, and a signal can have multiple I/O pins allowing for external connections at different location when performing hierarchical layout.

Dense layout cannot typically be fully automated using a simple set of algorithms. Often user input is required to identify opportunities for optimization that, to date, have eluded automated approaches. To support this, `pandR` has a number of commands that can be used to modify any layout and use automated support only when it is appropriate. In this sense `pandR` can be thought of as a much more sophisticated layout editor where the primitive operations are the same as a tool such as Virtuoso from Cadence. The difference is that the tool also includes support for automated stack generation, placement, and routing and a user can pick and choose what automation support they desire.

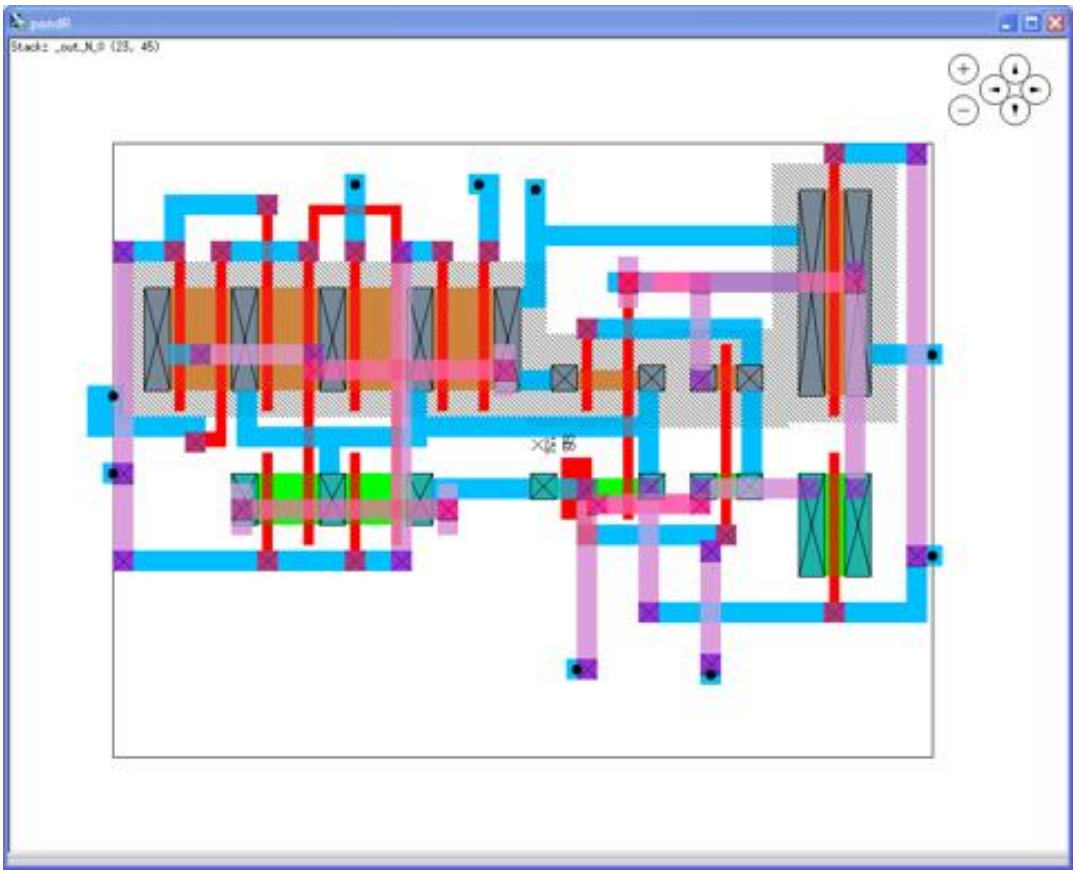

Figure 10. Screenshot of the pandR layout assistant.

An important feature of `pandR` is that all commands and actions of a user of the tool are captured in a script. This script can be "re-played" to re-create the physical layout. This replay procedure is deterministic, and captures a mixture of command-line operations as well as interactive commands (for example, drag and drop of part of the layout). Every operation that modifies the layout is captured in a file so that the layout creation process is reproducible.

For more complex circuits that are hierarchical, the tool is able to layout each sub-circuit individually and then combine those layouts into a higher level module. The sub-circuits are "grey boxed" and placed as individual components, whereas wire routing is aware of the internal geometry of the sub-circuit.

Figure 11 shows a screenshot of the physical layout of the AFPGA. The tile consists of two sections that are approximately equal in size. The left hand side of the tile corresponds to the internals of the logic block, including the lookup tables, arithmetic functions, and internal tile routing. The right hand side of the tile consists of the pipelined

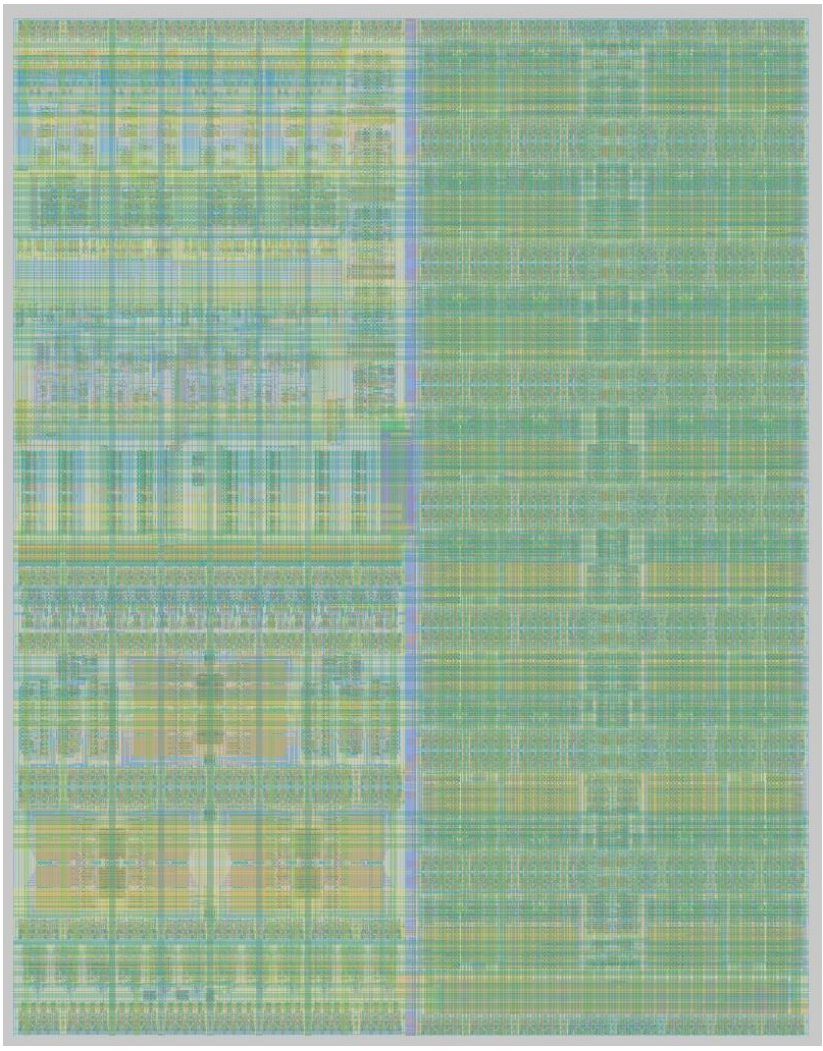

Figure 11. Physical layout of an arrayable tile of the new AFPGA.

interconnection network for intra-tile routing. The area of the tile used by logic functions is ≈10%, with configuration memory using ≈30% of the area and the rest being used by the routing network.

There were a number of design issues that required detailed analysis of analog effects by circuit simulations that modeled parasitic resistances and capacitances. While large simulations that contain accurate models are beyond the scope of existing commercial tools, simplified but conservative models were used to ensure robust functionality of the final AFPGA. Many changes to the underlying circuits were made based on these simulations, including partitioned wires for the configuration memory and appropriate spacing and shielding for the interconnect lines in the AFPGA.

Extensive SPICE simulations predict that the peak performance of this AFPGA is in the range of 800 MHz. This frequency was chosen based on the performance of the rest of the system (below 150 MHz) that will include the AFPGA fabric. The total number of AFPGA tiles in the system can be changed easily by simply creating an array of the appropriate size.

The edge of the tile contains interface circuits as well as programming support for the configuration memory. This interface exports a synchronous interface to the rest of the system for easy integration with commercial physical design flows. With help from Prof. Stine of Oklahoma State University, a flow was created that enabled the entire physical AFPGA block to resemble a synchronous IP block that was placed and routed with Cadence Encounter.

In collaboration with AFRL, the final physical implementation was submitted for fabrication through the Trusted Foundry Access Program to IBM's foundry services.

4. Summary

Previous research on asynchronous FPGA architectures at Cornell resulted in the development of a new high performance reconfigurable fabric. This funded effort transitioned the new technology to a sensitive AFRL project that combined general-purpose computing and reconfigurable logic for cognitive information processing applications. In a joint development effort, a large 65nm chip was created with AFRL contributing portions of the design and Cornell contributing a high-performance asynchronous FPGA. For suitability for cognitive operations, the AFPGA was designed to support dynamic reconfiguration and virtualization of its resources. Additional features were introduced to enable the AFPGA configuration to be trustworthy even if the system design was compromised. A chip to demonstrate the new architecture was submitted for fabrication through the Trusted Foundry Access Program. A follow on project was begun to update the design tools for a 32nm fabrication process design flow and add new security features for inclusion in a future design.

5. References

[1] J. Martinez, C. Gomes, R. Linderman. Research Gap Analysis Report. *Workshop on Research Directions in Architectures and Systems for Cognitive Information Processing*, July 2005.

[2] John Teifel and Rajit Manohar. Programmable Asynchronous Pipeline Arrays. *Proceedings of the 13th International Conference on Field Programmable Logic and Applications*, Lisbon, Portugal, September 2003.

[3] John Teifel and Rajit Manohar. Highly Pipelined Asynchronous FPGAs. *12th ACM International Symposium on Field-Programmable Gate Arrays*, Monterey, CA, February 2004.

[4] David Fang, John Teifel, and Rajit Manohar. A High-Performance Asynchronous FPGA: Test Results. *2005 IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2005.

[5] Xilinx. VirtexTM 2.5V field programmable gate arrays. Xilinx Data Sheet, 2002.

[6] J. B. Dennis. The evolution of 'static' data-flow architecture. In J.-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*. Prentice-Hall, 1991.

[7] Song Peng, David Fang, John Teifel, and Rajit Manohar. Automated Synthesis for Asynchronous FPGAs. *13th ACM International Symposium on Field Programmable Gate Arrays*, February 2005.

[8] I. Kuon, R. Tessier, J. Rose. FPGA Architectures: Survey and Challenges. *Foundations and Trends in Electronic Design Automation*, Vol. 2, No. 2, pp. 135-253, 2007.

[9] John Teifel and Rajit Manohar. Static Tokens: Using Dataflow to Automate Concurrent Pipeline Synthesis. *Proceedings of the 10th International Symposium on Asynchronous Circuits and Systems*, April 2004.

[10] Jon Russo, Mohammed Amduka, Keith Pendersen, Richard Lethin, Jonathan Springer, Rajit Manohar, Rami Melhem. Enabling Cognitive Architectures for UAV Mission Planning. *Proceedings of the High Performance Embedded Computing Workshop*, September 2006.

[11] I. Kuon and J. Rose. Measuring the Gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems*, Vol. 26, No. 2, February 2007.

[12] Cadence Design Systems. Clock Domain Crossing: Closing the Loop on Clock Domain Functional Implementation. Technical paper, 2004.

[13] R. Ginosar. Fourteen ways to fool your synchronizer. *Proceedings of the IEEE International Symposium on Asynchronous Circuits and Systems*, May 2003.

[14] R. Manohar. Systems and methods for performing automated conversion of representations of synchronous circuit designs to and from representations of asynchronous circuit designs. US Patent 7,610,567, October 2009.

[15] B. Hu, M. Marek-Sadowska. Multilevel fixed-point-addition-based VLSI placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 24, No. 8, 2005.

6. Acronyms

Acronym	Expanded Form
ACT	Asynchronous Circuit Tools
AFPGA	Asynchronous Field Programmable Gate Array
AFRL	Air Force Rome Laboratories
CB	Connection box
DARPA	Defense Advanced Research Projects Agency
EDA	Electronic Design Automation
FPGA	Field Programmable Gate Array
I/O	Input/Output
IP	Intellectual Property
LB	Logic block
LUT	Lookup Table
SB	Switch box
SRAM	Static Random Access Memory
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLSI	Very Large Scale Integration